NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE RELIABILITY
MANAGEMENT THROUGH METRICS

by

Douglas R. Burton

March 1993

Thesis Advisors: Donald P. Gaver
Norman F. Schneidewind

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1993	Master's Thesis

4. TITLE AND SUBTITLE
Software Reliability Management Through Metrics

5. FUNDING NUMBERS

6. AUTHOR(S)
BURTON, Douglas R.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release, distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As systems become increasingly software dependent, their reliability will accordingly depend more so on the reliability of their resident software. Just as techniques and processes were developed and improved to ensure hardware reliability, so must techniques evolve to ensure software reliability.

Two questions are addressed by this thesis. First, how do we measure software reliability throughout a project's lifecycle? Second, is there a tool which will provide effective insight into the test-now-or-later problem?

The solution to our first question is a U.S. Army software procurement methodology which is briefly outlined as the overall framework for software procurement in this thesis. A software fault analysis tool is developed and programmed. Some results from this algorithm are provided and their potential resource saving impact explored.

Program managers of software-intensive projects would be well advised to use the Army's methodology and this fault analysis tool to potentially save critically short procurement resources.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Reliability, Software Metrics, Reliability	70
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited.

SOFTWARE RELIABILITY
MANAGEMENT THROUGH METRICS

by

Douglas R. Burton
Lieutenant, United States Navy
B.S. Aerospace Engineering, U.S. Naval Academy, 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
March 1993

Author: _____
Douglas R. Burton

Approved by: _____
Donald P. Gaver, Thesis Advisor

_____
Norman F. Schneidewind, Thesis Advisor

_____
Patricia A. Jacobs, Second Reader

_____
Peter Purdue, Chairman
Department of Operations Research

ABSTRACT

As systems become increasingly software dependent, their reliability will accordingly depend more so on the reliability of their resident software. Just as techniques and processes were developed and improved to ensure hardware reliability, so must techniques evolve to ensure software reliability.

Two questions are addressed by this thesis. First, how do we measure software reliability throughout a project's lifecycle? Second, is there a tool which will provide effective insight into the test-now-or-later problem?

The solution to our first question is a U.S. Army software procurement methodology which is briefly outlined as the overall framework for software procurement in this thesis.

The solution to our second problem is the primary focus of this thesis. A software fault analysis tool is developed and programmed. Some results from this algorithm are provided and their potential resource saving impact explored.

Program managers of software-intensive projects would be well advised to use the Army's methodology and this fault analysis tool to potentially save critically short procurement resources.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

I. INTRODUCTION

As the Department of Defense progresses through the end of the twentieth century and into a new and very uncertain twenty-first century, one area can be considered a near certainty: system reliability will become increasingly dependent on software execution. Today many systems (Patriot anti-missile, Space Shuttle, weapons' guidance, and aircraft control systems among others) are being deployed which depend extensively on correct software execution in order to achieve desirable and intended overall system task execution. Successful task execution is essential for overall mission success and goal achievement; misperformance of even a simple computing task can result in total mission failure. Mission failure could certainly mean returning to base because of a weapon's guidance error, without destroying a single target, or it could mean failing to destroy an incoming ballistic missile, as a result of an imperceptible guidance error, with disastrous consequences. Such outcomes are unacceptable, and quite fortunately, occur very infrequently.

Experience with modern systems during Operations Desert Shield and Desert Storm indicates that the Department of Defense system procurement policies generally produce systems of acceptable quality and suitability. Suitability is a clan of system properties that includes reliability, defined to be

the probability that the system performs its mission without functional failure. Since modern weapons, such as missiles, depend upon electronic sensors and information-processing components to carry out their mission, reliability of the sophisticated electronics and its software is essential to achieve mission success.

To produce highly reliable products many system tests, analyses, and simulations are typically conducted before release is granted to actual operators. This extensive process consumes many procurement resources (man-hours, platform availability hours, and dollars). Errors, resulting from faulty design, inadequate simulation, and premature system test during the procurement process result in the waste of valuable resources. In the light of nearly guaranteed reductions in procurement resource availability in years to come, and the ever-increasing dependency of systems on software for task execution, the Department of Defense must strive to procure and test systems economically and in minimal time.

Managing the procurement process is no simple task, and there are gray-areas in which mistakes will often be made. However, a tool, or tools, which potentially reduces the number of these gray-area mistakes will certainly reduce the consumption of procurement resources. This reduction in resource consumption by individual projects, accompanied by a

constant or increased level of system reliability and performance, becomes the procurement manager's goal.

This thesis will address an approach to assist in achieving the procurement manager's goal that uses software metrics as statistical indicators of the attained level of software reliability of a system. A manager responsible for procurement and acceptance of software intensive systems is well-advised to use these metrics to help make critical test-now-or-later decisions. The guidance offered may result in significant resource savings if it leads to appropriately postponing a prematurely scheduled test, one highly likely to lead to early termination without obtaining useful results, but which, nevertheless, consumes many man-hours, much platform time, and many dollars. A well-supported decision as to when software intensive systems are ready for operational test potentially saves both project time and money.

In the next two sections a two-part software reliability problem will be described. The overall problem of improving the software procurement process and the subset problem of gray-area test-now-or-later decisions will be included. In Chapter II, a metrics-supported solution will be presented to address the overall problem. In Chapter III, a specific fault analysis model will be described to address the gray-area decision problem. In Chapter IV, two sets of software fault data are analyzed using an algorithm which represents the fault analysis model from Chapter III. In Chapter V,

conclusions and recommendations are presented. The appendices include: details about 12 base software metrics from Chapter II, the algorithm based on the Chapter III fault analysis model, and an additional fault-analysis model which is slightly different from the one in Chapter III.

A. PROBLEM BACKGROUND

Since approximately 1970, software reliability has steadily emerged as a primary area of interest for procurement agencies tasked with purchasing software-intensive systems. Both civilian and Department of Defense procurement agencies acquire software-intensive systems regularly.

In modern times measurement techniques have been developed and integrated into the hardware acquisition process. For example, an M-16 semi-automatic assault rifle goes through an extensive process of measurement and re-measurement in order to confidently ascertain that a very reliable weapon is delivered to the soldier in the field.

At present, many hardware items depend in an important way on sister software units for information processing and guidance to comply with documented requirements. Such dependency upon software can be anticipated to increase over time. For example, the software interface and control for a Vulcan Phalanx anti-air defense cannon is one such sister software unit, the failure of which can result in catastrophic

system failure, or, at the very least, in considerable inconvenience.

B. THE TWO-PART PROBLEM

The assessment of software reliability, while consuming minimal procurement resources, becomes the overall problem addressed by this thesis. A software metric or measurement process is provided in Chapter II to address this overall software reliability problem. Over time, progress has been made in developing software performance measurement techniques that are increasingly cost-effective. These techniques are still in their infancy, yet do, and have, frequently achieved their objectives.

An example of this progress in civilian-sector practice is the current process of software performance measurement practiced at IBM, Houston, for the Space Shuttle's primary avionics software (Keller, 1992). The measurement tool used at IBM is a Schneidewind non-homogeneous Poisson fault analysis model (Farr, 1991, Schneidewind, 1992). The objective of this IBM measurement tool is to predict the probability of encountering a serious primary software error during onboard processing on the next Shuttle mission. This objective falls squarely in line with the overall Shuttle project manager's goal of preventing any such failures. Thus, with this measurement tool the Shuttle project manager could reasonably expect to make appropriate decisions concerning the

level and extent of ground testing, which would directly result in resource savings.

To illustrate our second problem concerning gray-area decision mistakes and their ramifications in defense acquisition, a potential scenario (U.S. Navy-specific) is appropriate. Suppose that an anti-air, surface-launched missile system depends heavily on its resident software to receive, interpret and execute all applicable sensor, operator, and system inputs. Suppose this missile system's program manager, without software metric information, feels that an operational system test is now warranted for this missile system. In conjunction with COMOPTEVFOR (Commander, Operational Test and Evaluation Forces), the missile system is readied for test. So, with a warm feeling about the missile system, the project manager pushes the system through to this operational testing phase, very possibly a marginally-justifiable or gray-area decision. At this point, during the OP-test, a severe software fault could result in significant resource losses (e.g., many man-hours, much platform time, and monies). To reduce the probability of this resource-wasting scenario occurring, a fault-analysis model is provided in Chapter III as a specific tool to potentially reduce the number of these gray-area mistakes. As part of the overall software reliability process proposed in Chapter II, a predictive software fault analysis is developed in this thesis to address this second problem. Previous research, entitled

Fitting and Prediction Uncertainty for a Software Reliability Model (Dennison, 1992), has been undertaken. In this previous study a Non-Homogeneous Poisson Process (NHPP) model was presented and analyzed. This thesis will develop a generalized version of this previous work which utilizes a different likelihood approximation which allows for varying software execution times covering separate operating periods (i.e., weekly, daily, etc.). This generalized version supports data akin to most real-world periodic data collection techniques.

Our problem then has been presented in two parts. Stated as questions: first, is there an approach whereby software reliability can be increased to operationally acceptable levels throughout the procurement process, and second, prior to the operational test phase, can we use a specific tool to predict the propensity for errors to occur in the near future for our software-intensive system? Solutions to these problems will be addressed in the next few chapters.

In Chapter III, a software-metrics approach will be introduced and discussed as a positive solution to our first problem. Also, a metric quality assurance (e.g., appropriate metric selection) question will be raised and discussed.

II. A METRIC SOLUTION FOR SOFTWARE MANAGEMENT

The Department of the Army has put together an extensive software development guidance package draft entitled Software Test and Evaluation Procedures and Guidelines, (Draft Software, 1992). In this document, a process to economically procure defense software is thoroughly addressed. The basis for this effective process is continuous evaluation. "Decision-making must be based upon substantive evaluations of software characteristics, maturity, and reliability indicators throughout the lifecycle" (Army, 1-5, 1992). With this goal the Army plan uses software metrics, which are defined as "a quantitative value, procedure, methodology, and/or technique which allows one the ability to measure various aspects and characteristics of software," (Army, Glossary-5, 1992) to quantify software characteristics so that a program manager can indeed make decisions based on the metric information which will either save procurement resources, produce more capable systems, or both.

This draft process directly addresses our first and overall software reliability problem. A group of 12 primary software metrics form the building blocks for the Army's proposed metric analysis for software-intensive systems. These 12 metrics are outlined in Appendix A and briefly discussed in this chapter. The Department of the Army also

places each of these 12 metrics into one of three categories; *management*, *requirements* or *quality*.

The *management* category includes four metrics: *cost*, *schedule*, *computer resource utilization* and *software engineering environment*. *Cost* is a tool to compare budgeted versus actual costs as well as scheduled versus actual progress. *Schedule* provides insight into milestone progress, or lack thereof. *Computer resource utilization* is a tool to track the degree of computer processing usage (e.g., computer processing units, input processors, and memory registers). The *software engineering environment* metric is a numerical rating tool for any given contractor, based on evidence of that contractor's historical adherence to certain software engineering practices and procedures.

The *requirements* category contains both *requirements traceability and stability* metrics. *Traceability* provides a measure of contractor conformity to system requirements (e.g., the percentage of system requirements that are being met by the software code). *Stability* indicates how much change the system requirements have exhibited because of software non-conformity (e.g., the percentage of requirement changes from the baseline initial project list of requirements).

Lastly, the *quality* category contains six software metrics: *design stability*, *complexity*, *breadth of testing*, *depth of testing*, *fault profile*, and *reliability*. *Design stability* reflects the amount of change to software design

(e.g., percentage of the source line of code (SLOC) that is affected by any implemented change to the software design). *Complexity* supplies insight into the structure of the software and includes measures, like the McCabe cyclomatic complexity measure (McCabe, 1976), to quantify the structure for any software module. *Breadth of testing* indicates through percentage conformity how well testing has covered the functional requirements. *Depth of testing* helps indicate the extent that executable paths within individual modules have been exercised (e.g., percentage of the executable paths utilized). *Fault profiles* furnish insight into a contractor's ability to correct known discrepancies, and provide rough insight into software quality through fault correction tendencies and fault occurrence tracking. Finally, the *reliability* metric utilizes modeling techniques to make predictions of future software readiness, meaning freedom from faults during execution.

These 12 base metrics are each detailed further in Appendix A. Example figures and applicable equations are also included as part of the appendix.

To utilize these 12 metrics as an effective managerial tool, a program manager would consider all available metric information as a part of any milestone review process. Each applicable metric can provide the additional information necessary to help signal problems or progressions within the software's development.

At each milestone review the software-metric information should be made available to all involved agencies. This information-sharing policy would potentially lead to earlier software problem identification which should in-turn result in resource-saving decisions. For example, a significant software inadequacy relating to contractor requirements accomplishment would certainly be indicated in the software *requirements traceability* and *stability* metrics information as numerically low conformity percentages. This requirements inadequacy would in all likelihood surface earlier in the acquisition process given the software-metrics information. This quantitative and often earlier problem recognition would most certainly lead to timely managerial attention and resulting resource savings. Thus, the metrics methodology becomes a quantitative management tool for any software-intensive program.

The 12 Army metrics form a base set of software metrics within which deletions or additions may be appropriate. Many other metrics exist which may or may not be appropriate for a particular project (Siefert, 1989). To start with this set of 12 metrics as the primary initial evaluation point becomes an initial and basic goal for a project manager. From this initial set, some metrics may be considered inapplicable and consequently removed from consideration for inclusion in project data requirements. Still others, outside the basic 12, may be incorporated into the list of applicable metrics,

and be subsequently included in the project data requirements list.

Given the Army metric plan, the selection and evaluation of appropriate software metrics is no simple endeavor, but is essential for obtaining useful quantitative information throughout the project lifecycle. Guidance to choose suitable software metrics would be beneficial as part of the software metric selection and utilization process. The recently approved and distributed IEEE Standard for a Software Quality Metrics Methodology (IEEE, 1993), promises to be a useful tool to aid in the evaluation and selection of an appropriate set of software metrics. Direct guidance for software metric validation is included as part of the process by which better overall software quality is achieved.

Once a set of software metrics has been selected and validated with respect to applicability to the system under development, and has been included in system data requirements, then, and only then, can a project manager expect to successfully receive, analyze, apply, and profit by information that he or she would obtain from the metrics throughout a project's lifecycle. Better software decisions (i.e., decisions that would lead to increased quality and expedited operational deployment) should evolve from an effectively managed software metric plan which would in turn lead directly to resource savings.

The software metric approach, when implemented as part of the milestone review process, can become a crucial management tool for all phases of software development. A prevalent problem appears to stem from the propensity of program managers to make premature resource-wasting decisions concerning the progression of their projects through the procurement process because of inadequacy of information concerning the faults currently resident in their software. The use of software metrics, after validation, should positively influence this decision making process at every stage of system procurement.

With the Army's process to address overall software management guidance, our second and much more specific problem can be addressed. This test-or-do-not-test decision problem is one that a program manager in conjunction with the testing agencies must make very often. A quantitative indicator which provides insight into the propensity for software to encounter future faults would be very valuable. With this obstacle in mind, a predictive tool for just this purpose is described and exercised in the next two chapters. The major impact of this thesis stems from Bootstrapping (Efron, 1985) a NHPP model (Goel, 1979) and a computer program that allows this Bootstrapped model to be applied to real test data.

III. RELIABILITY MODEL DEVELOPMENT

A Non-Homogeneous Poisson Process (NHPP) software reliability model is constructed using the following assumptions:

- Program test runs are conducted in non-overlapping time intervals.
- The fault counts in each interval are independent.
- Each fault is corrected as it is detected.
- The fault detection rate is modeled as exponentially decreasing over time.
- The faults each have the same severity.
- The faults are equally likely to be detected.
- Fault counts are recorded after like intervals of calendar time (e.g. daily, weekly etc.).

The mathematical model (Goel, 1979) is simply

$$P\{f_i | t_i\} = \frac{(e^{-\lambda_i} \cdot \lambda_i^{f_i})}{f_i!}, \qquad (1)$$

where the subscript i represents the ith run-time interval. These i intervals need not be of the same time length nor contain the same number of faults. They may represent the total time a program is run on a day, week, etc. The f_i term is the number of faults that occur during period i; t_i is the cumulative amount of program run time up to and including

interval i; and λ_i is the mean fault parameter for interval i. The parameter λ_i is further defined as

$$\lambda_i=\lambda\cdot[(1-e^{-\mu\cdot t_i})-(1-e^{-\mu\cdot t_{i-1}})] \tag{2}$$

so

$$\lambda_i=\lambda(e^{-\mu\cdot t_{i-1}}-e^{-\mu\cdot t_i}). \tag{3}$$

In the present model λ and μ are the overall NHPP system parameters and are to be estimated using the data set. The maximum likelihood method (Larson, 1982) is used to estimate these parameters.

$$L(\lambda,\mu|data)=\prod_{i=1}^{I} e^{-\lambda_i}\cdot\frac{(\lambda_i)^{f_i}}{f_i!} \tag{4}$$

is the likelihood function. Taking the natural log and using I to represent the total number of time intervals results in

$$\bar{L}(\lambda,\mu|data)=\ln L(\lambda,\mu|data)=-\sum_{i=1}^{I}\lambda_i+\sum_{i=1}^{I} f_i\cdot\ln\left(\frac{\lambda_i}{f_i!}\right). \tag{5}$$

Substituting the expression for λ_i from (3) we get

$$\bar{L}(\lambda,\mu|data) = -\sum_{i=1}^{I}\lambda\cdot(e^{-\mu\cdot t_{i-1}}-e^{-\mu\cdot t_i})+\sum_{i=1}^{I}f_i\cdot\ln[\lambda] \qquad (6)$$
$$+\sum_{i=1}^{I}f_i\cdot\ln[e^{-\mu\cdot t_{i-1}}-e^{-\mu\cdot t_i}]-\sum_{i=1}^{I}f_i\cdot\ln(f_i!)\,.$$

Next, to find a maximum for this log-likelihood function take the partial derivative of $\bar{L}(\lambda,\mu|\text{data})$ with respect to λ to get

$$\frac{d\bar{L}}{d\lambda}=-\sum_{i=1}^{I}(e^{-\mu\cdot t_{i-1}}-e^{-\mu\cdot t_i})+\sum_{i=1}^{I}\frac{f_i}{\lambda}\,. \qquad (7)$$

Sum out (7) to obtain

$$\frac{d\bar{L}}{d\lambda}=(1-e^{-\mu\cdot t_T})+(\frac{f_+}{\lambda})\,, \qquad (8)$$

where f_+ is the cumulative number of faults for all periods i. Equate the derivative to zero and solve for the maximum likelihood estimate

$$\hat{\lambda}=\frac{f_+}{(1-e^{-\mu\cdot t_T})}\,. \qquad (9)$$

After replacing λ in (6) with $\hat{\lambda}$ from (9) take the derivative of the log-likelihood (6) with respect to μ and set this equal to zero, again to obtain a maximum likelihood estimate, to get

$$\sum_{i=1}^{I} \left[\frac{f_+ \cdot (e^{-\mu \cdot t_{i-1}} - e^{-\mu \cdot t_i})}{(1-e^{-\mu \cdot t_I})} - f_i\right] \cdot \left[\frac{-t_{i-1} \cdot e^{-\mu \cdot t_{i-1}} + t_i \cdot e^{-\mu \cdot t_i}}{e^{-\mu \cdot t_{i-1}} - e^{-\mu \cdot t_i}}\right] = 0. \quad (10)$$

To solve for μ in (10) an iterative procedure is used. First, executing the summation in (10) results in

$$\frac{f_+ \cdot t_I \cdot e^{-\mu \cdot t_I}}{1-e^{-\mu \cdot t_I}} - \sum_{i=1}^{I} f_i \cdot \left[\frac{-t_{i-1} \cdot e^{-\mu \cdot t_{i-1}} + t_i \cdot e^{-\mu \cdot t_i}}{e^{-\mu \cdot t_{i-1}} - e^{-\mu \cdot t_i}}\right] = 0. \quad (11)$$

Let $x=(\exp(-\mu \cdot t_I))$ and also let

$$s = \frac{1}{f_+ \cdot t_I} \sum_{i=1}^{I} f_i \cdot \left[\frac{-t_{i-1} \cdot e^{-\mu \cdot t_{i-1}} + t_i \cdot e^{-\mu \cdot t_i}}{e^{-\mu \cdot t_{i-1}} - e^{-\mu \cdot t_i}}\right]. \quad (12)$$

Now, with (11) and (12) we observe that $s=[x/(1-x)]$ and alternately $x=[s/(1+s)]$.

To begin the iterative process, x is set initially to

$$\left[\frac{\textit{number of faults for last half of the run time}}{\textit{number of faults for first half of the run time}}\right]^2. \quad (13)$$

This rough initial x-value is obtained from observing that the square of the fraction of the expected value of the numerator divided by the expected value of the denominator is indeed a potential x-value choice. This rough x-value need not be very precise. The iterative process will refine it quickly. Now, with $x=(\exp(-\mu \cdot t_I))$, solve for the first μ-value. Then with this μ-value solve (12) to get s. We then obtain our next x-value from $x=[s/(1+s)]$, which iteratively leads to a series of μ-values that are eventually an accordingly small number, epsilon equal to 10^{-3}, apart. When this is achieved the most

recent μ-value becomes our model estimated parameter, $\hat{\mu}$. From (9) we get our estimate for λ which is our other model parameter, $\hat{\lambda}$.

With the estimates $\hat{\mu}$ and $\hat{\lambda}$, obtained from the raw data, we can now proceed with the Bootstrapping technique (Efron, 1985). This procedure recognizes the variability of raw data, and by re-sampling allows us to expand our analysis from a point estimate $(\hat{\mu},\hat{\lambda})$ to confidence intervals.

First, using (3) and the estimates of λ and μ from the original data, an estimated λ_i is computed for each interval i. These λ_i's are then used to generate a new set of synthetic observed data, denoted $f_i(b)$, using a Poisson random number generator (Fishman, p. 440, 1978). Once this new data set is available, a series of new estimates $\hat{\mu}(b)$ and $\hat{\lambda}(b)$ are obtained using the same procedures as above: (9) and (10). This data regeneration and estimation procedure is repeated 200 times, giving $(\hat{\mu}(b), \hat{\lambda}(b))$ for b=1, 2, 3,..., 200; our pairs of bootstrapped estimates.

The resulting set of 200 estimated parameter sets forms the nucleus for our statistical analysis, to be described below.

Suppose there is interest in estimating the expected number of faults to occur in a run-time interval of duration τ following the current period, (e.g., after the total observed run-time t_l has elapsed). In theory this is

$$m(\tau; t_I; \lambda, \mu) = E[N(t_I+\tau) - N(t_I)] = \lambda \cdot e^{-\mu \cdot t_I} \cdot (1 - e^{-\mu\tau}) . \qquad (14)$$

Now the maximum likelihood estimate (mle) of this mean number of faults is obtained by substituting in the values of the mle for the parameters λ and μ. Thus

$$\hat{m}(\tau; t_I; \lambda, \mu) = m(\tau; t_I; \hat{\lambda}, \hat{\mu}) = \hat{\lambda} \cdot e^{-\hat{\mu} \cdot t_I} \cdot (1 - e^{-\hat{\mu}\tau}) . \qquad (15)$$

Confidence limits can be placed on the mean number of faults, and on the probability distribution of the number of future faults, by using the bootstrapped parameter estimates referred to earlier. Here is the procedure.

(a) Evaluate

$$\hat{m}(b) \equiv m(\tau; t_I; \hat{\lambda}(b), \hat{\mu}(b)) \quad \textit{where } b=1,2,\ldots,B. \qquad (16)$$

Note that the bth bootstrap estimate occurs in the estimated mean.

(b) Sort these values in increasing order:

$$\hat{m}_1 < \hat{m}_2 < \hat{m}_3 < \ldots < \hat{m}_B, \qquad (17)$$

where $(\hat{m})_j$ is the jth ordered value of $(\hat{m}(b))$, b=1, 2, 3, ..., B.

(c) Define $j(\alpha) = [\alpha B]$ = smallest integer at least as large as αB, where $\alpha \cdot 100$ is the desired percent confidence

(e.g., α=0.95). Then quote the ordered value $\hat{m}_{j(\alpha)}$ as the approximate one-sided $\alpha \cdot 100\%$ confidence limit for the mean number of faults in interval $(t_I, t_I+\tau)$ (e.g., with confidence $\alpha \cdot 100\%$ the mean number of faults in that interval is less than or equal to $\hat{m}_{j(\alpha)}$. For this thesis upper (α=0.95) and lower (α=0.05) one-sided confidence intervals are studied.

(d) Because the distribution function of the Poisson is specified by its mean we can state that with confidence $\alpha \cdot 100\%$ the probability that there will be no more than k faults appearing in $(t_I, t_I+\tau)$ is

$$\sum_{i=0}^{k} \frac{e^{-\hat{m}_{j(\alpha)}} \cdot (\hat{m}_{j(\alpha)})^i}{i!}; \qquad (18)$$

in particular with confidence approximately $\alpha \cdot 100\%$ the probability of zero faults is $\exp(-\hat{m}_{j(\alpha)})$.

These model results become one basis upon which the program manager and the testing agency can help determine whether a project's software is indeed mature enough to support further testing or deployment.

If the precise inter-occurrence run-times are known then a slightly different NHPP formulation for the maximum likelihood estimates of λ and μ are [illegible]ed. These are presented in Appendix C.

This NHPP model along with measured run-times and faults recorded weekly or daily, etc., form the basis for a useable prediction algorithm which is included as Appendix B. The results obtained from using this algorithm on two separate data sets are presented in Chapter IV.

IV. MODEL APPLICATION

Two available sets of data (COMOPTEVFOR, 1992) were used to exercise the NHPP algorithm (Appendix B). These two data sets are provided in Tables 1 and 2 below. The data are displayed as the amount of software execution time during each week long period and the associated number of software faults encountered during that same week. Notice that Data Set One's execution times are recorded in minutes, and Data Set Two's in hours.

TABLE 1. DATA SET ONE

WEEK	TIME IN MINUTES	FAULT COUNTS
1	5.85	6
2	5.97	5
3	19.38	8
4	42.70	9
5	10.92	4
6	21.51	8
7	25.70	8
8	51.00	7
9	26.29	5
10	63.30	12

TABLE 2. DATA SET TWO		
WEEK	TIME IN HOURS	FAULT COUNTS
1	40	3
2	30	7
3	67	4
4	35	1
5	38	6
6	30	6
7	25	1
8	25	1
9	85	4
10	41	3

For algorithm execution, the data sets above were entered as cumulative execution times and cumulative fault counts.

Each of the two data sets contain ten data points. Starting with only the first five data points, two predictions were made using the NHPP algorithm. First, a prediction utilizing five data points over the next two weeks (i.e., end of the seventh week) of software execution time was made. Second, a prediction utilizing the same five data points over all available data points (i.e., end of the tenth week) was made. These, two week, and, end of data, predictions were also calculated for six, seven, eight, and nine data points. Results are provided in Tables 3 and 4 for Data Set One and in Tables 5 and 6 for Data Set Two. Data in Tables 3, 4, 5, and 6 is for 95% one-sided confidence intervals.

TABLE 3. DATA SET ONE
TWO WEEK PREDICTIONS (NEAR TERM)
95% ONE-SIDED LOWER AND UPPER CONFIDENCE INTERVALS
PREDICTED MEAN NUMBER OF FAULTS

NUMBER OF DATA POINTS	EXECUTION TIME (MINUTES)	ACTUAL NUMBER OF FAULTS	95% LOWER	95% UPPER
5	47.21	16	5.02	11.93
6	76.70	15	13.44	25.30
7	77.29	12	14.54	24.65
8	89.59	17	12.97	24.06
9	NA	NA	NA	NA

TABLE 4. DATA SET ONE
END OF DATA PREDICTIONS (LONG TERM)
95% ONE-SIDED LOWER AND UPPER CONFIDENCE INTERVALS
PREDICTED MEAN NUMBER OF FAULTS

NUMBER OF DATA POINTS	EXECUTION TIME (MINUTES)	ACTUAL NUMBER OF FAULTS	95% LOWER	95% UPPER
5	186.80	40	8.87	24.65
6	165.29	32	20.94	43.18
7	139.59	24	23.02	41.85
8	89.59	17	12.97	24.06
9	63.30	12	10.71	19.63

TABLE 5. DATA SET TWO
TWO WEEK PREDICTIONS (NEAR TERM)
95% ONE-SIDED LOWER AND UPPER CONFIDENCE INTERVALS
PREDICTED MEAN NUMBER OF FAULTS

NUMBER OF DATA POINTS	EXECUTION TIME (HOURS)	ACTUAL NUMBER OF FAULTS	95% LOWER	95% UPPER
5	55	7	4.53	9.05
6	50	2	5.18	10.59
7	110	5	6.88	13.82
8	126	7	2.94	10.88
9	NA	NA	NA	NA

TABLE 6. DATA SET TWO
END OF DATA PREDICTIONS (LONG TERM)
95% ONE-SIDED LOWER AND UPPER CONFIDENCE INTERVALS
PREDICTED MEAN NUMBER OF FAULTS

NUMBER OF DATA POINTS	EXECUTION TIME (HOURS)	ACTUAL NUMBER OF FAULTS	95% LOWER	95% UPPER
5	206	15	13.81	27.61
6	176	9	5.99	11.99
7	151	8	6.97	13.96
8	126	7	2.94	10.88
9	41	3	2.78	4.96

From Tables 3 through 6, one can see with both data sets that as the number of data points available to the NHPP algorithm increases, the mean fault confidence limits tend to surround the actual fault counts more frequently. Also, for predictions to the end of the data set, the lower and upper confidence levels for Data Set One using only five data points were much lower than the fault counts observed: there were 40

actual faults, but the lower and upper confidence limits were 8.87 and 24.65, respectively. The discrepancy suggests the importance of caution at this stage.

Other interesting observations include:

- Four of ten short-term (i.e., two-week) predictions bounded the actual number of faults experienced.
- Total run-time for Data Set One was just over four and a half hours while Data Set Two had 416 total run-time hours.
- Data Set One had an overall fault rate of 15.85 faults per hour (immature software) while Data Set Two had an overall fault rate of 0.09 faults per hour (mature software).
- One-sided upper and lower confidence limits are closer to actual fault counts for Data Set Two than for Data Set One.

To briefly illustrate the tabular results of Tables 3, 4, 5 and 6, Figure 1, for Data Set One, and Figure 2, for Data Set Two, are provided. All figures produced in this thesis were generated using GRAFSTAT (GRAFSTAT, 1988), a commercial statistical analysis tool. In these two figures nine weeks or points of data were used with predictions provided over the tenth week of software execution. These figures reflect the nine data point results provided in the tables above. In practice all data points would be used and predictions would be calculated for a user-selected period of execution time

(e.g., the system software is required to operate for three hours in the next test phase, so a three-hour predictive window is selected by the user).

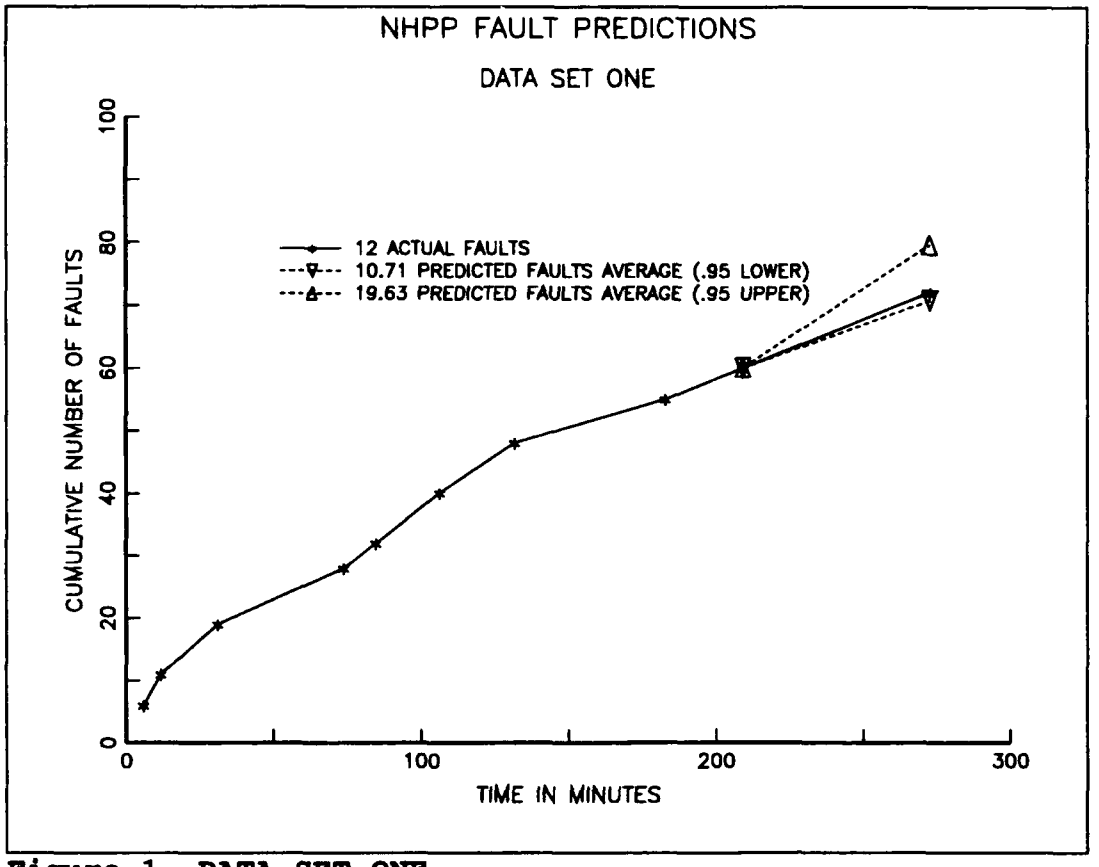

Figure 1 DATA SET ONE

Figure 1, from Data Set One, displays a lower one-sided 95% confidence prediction of 10.71 mean faults and an upper one-sided 95% confidence prediction of 19.63 mean faults. These predictions were made for an additional execution time of 63.3 minutes which coincides with the amount of execution time in the tenth week of data. There were 12 actual faults recorded in this same period. Thus our predicted mean values bound the actual number of faults recorded. A program manager, prior to

week ten, using this model and algorithm could have predicted that with 95% confidence this project's software would, on the average, have no fewer than 10.71 faults or no more than 19.63 faults in the next 63.3 minutes of software execution time. Additionally, with 90% confidence, this same manager could predict that the mean number of faults to occur in the next 63.3 minutes of software run-time would be between 10.71 and 19.63, accounting for th 5% upper and lower one-sided confidence tails.

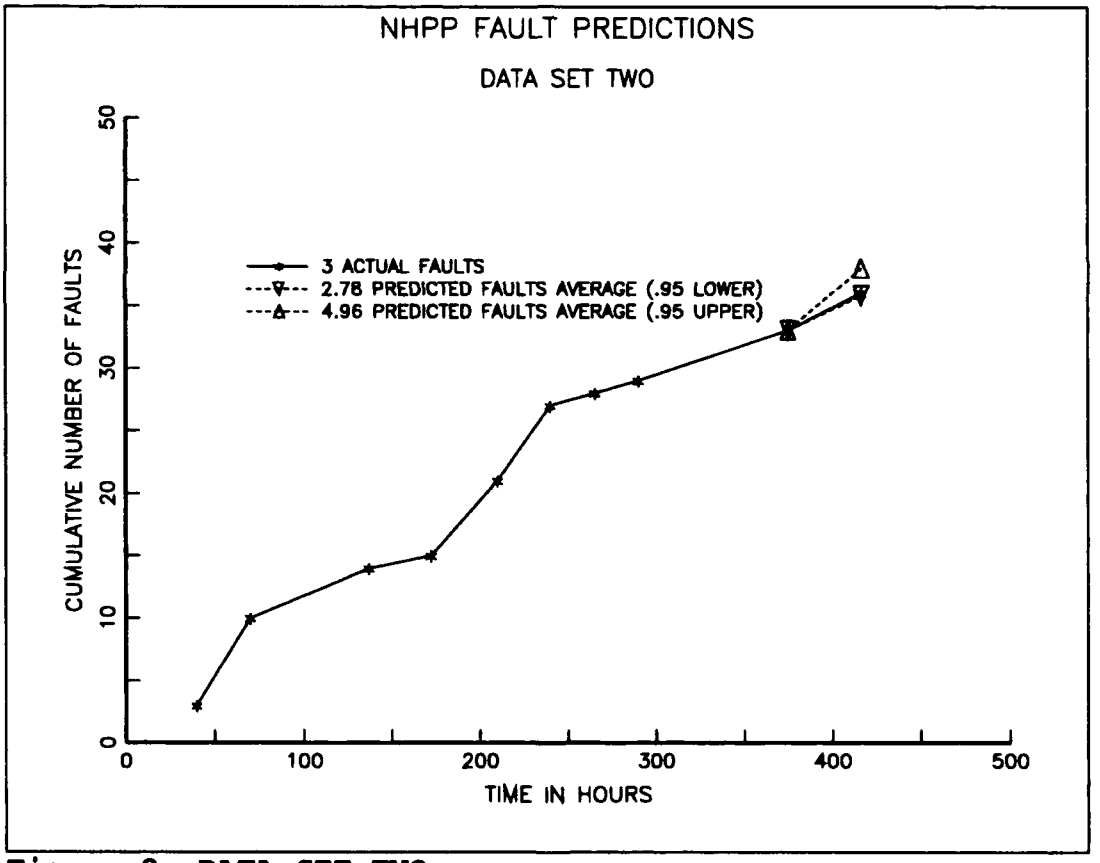

Figure 2 DATA SET TWO

Figure 2, from Data Set Two, displays a lower one-sided 95% confidence prediction of 2.78 mean faults and an

upper one-sided 95% confidence prediction of 4.96 mean faults. These predictions were made for an additional execution time of 41 hours which coincides with the amount of execution time in the tenth week of data. There were three actual faults recorded in this same period. Thus our predicted mean values bound the actual number of faults recorded. A program manager, prior to week ten, using this model and algorithm could have predicted that with 95% confidence this project's software would, on the average, have no fewer than 2.78 faults or no more than 4.96 faults in the next 41 hours of software execution time. Additionally, with 90% confidence this same manager could predict that the mean number of faults to occur in the next 41 hours of software run-time would be between 2.78 and 4.96, accounting for the 5% upper and lower one-sided confidence tails.

An additionally interesting prediction involves allowing for a very long prediction interval, approaching infinity. This prediction furnishes information about the number of faults remaining in the software and is included as Table 7. As expected from our NHPP model and plots of the data, many faults are likely to remain for both software projects when considering an infinite amount of software-execution time.

TABLE 7. BOTH DATA SETS FAULTS REMAINING PREDICTIONS (PRESENT TO INFINITY) 95% ONE-SIDED LOWER AND UPPER CONFIDENCE INTERVALS PREDICTED MEAN NUMBER OF FAULTS REMAINING				
NUMBER OF DATA POINTS	DATA SET ONE		DATA SET TWO	
	95% LOWER	95% UPPER	95% LOWER	95% UPPER
5	108	667	95	241
6	151	856	80	232
7	396	815	110	302
8	119	264	140	407
9	153	295	89	159

All of the results obtained indicate that this NHPP tool appears to be applicable to these data. Again, a rigorous validation process is required to ensure that applicable software metrics are selected for each individual software project.

In Chapter V, conclusions and recommendations concerning the overall metric process and this specific NHPP reliability model are presented.

V. CONCLUSIONS AND RECOMMENDATIONS

In light of the results from Chapter IV, it becomes apparent that a procurement agency would be well advised to implement a software metrics methodology. The Army plan will provide the overall structure to ensure that proper data is collected, and analysis conducted, during a software-intensive project's development and deployment.

Within the Army's software-metric methodology two other actions must also occur. First, the metric information must be made available to all involved agencies for analysis. One agency may realize something another has missed. This information sharing should result in an even greater resource savings. Secondly, each metric must be rigorously validated to ensure applicability within a project. Properly choosing the set of software metrics is the first and perhaps most important task in the resource saving metric process. Without these two actions, an otherwise good metrics methodology will perhaps not save as many procurement resources as it potentially might.

For software, the CRLCMP (Computer Resources Life-Cycle Management Plan), provides the instrument for implementing this metric process. Metric requirements and their inherent data requirements should be included in this document. It is recommended that the CRLCMP define clearly these metric and

data collection requirements so that each software-intensive project will, at inception, include adequate software metric data collection and analysis requirements.

From the results in Chapter IV, several conclusions can be drawn pertaining to the use of this NHPP algorithm.

First, and most importantly, a program manager could certainly benefit from the information obtained through this algorithm. For instance, from our data, after the sixth week for either system, a manager trying to decide whether or not to proceed with the next series of system tests, operational or developmental, would most assuredly have been better prepared to do so with the information calculated by this NHPP algorithm. The NHPP algorithm's injection of information quantifies the likely outcome of a test and its uncertainty as support for a forthcoming decision.

Second, the NHPP algorithm gave predictive results which frequently bounded the actual data results for both our immature (Data Set One) and mature (Data Set Two) software-intensive systems. These results suggest the applicability of the NHPP algorithm to quantitatively express the maturity (i.e., mature software contains relatively fewer faults) of a software-intensive system at all phases of its development (i.e., both early when systems are generally still immature, and later when such systems are generally more mature).

Third, when considering the long-term predictive results for either data set, a program manager could reasonably expect

to gain substantial insight into a project's milestone achievement potential. With this insight, a program manager could potentially save valuable procurement resources by making better decisions regarding software adjustments throughout the project's lifecycle.

Fourth, as more data were used to predict the expected mean number of faults in Chapter IV, the short-term predictive information became more accurate and the difference between the lower and upper 95% bounds became smaller. As data is accumulated a program manager could expect to receive more accurate short-term NHPP algorithm predictive results. This increased accuracy could reasonably lead to even better project software decisions.

Overall, this NHPP model with Bootstrapping can provide accurate and timely information to a program manager. The impact of this predictive model should result in better management decisions which in turn should result in the saving of valuable procurement resources. These savings would certainly pay dividends in DOD's procurement process.

When exercising this NHPP model it is recommended that SMERFS (Statistical Modeling and Estimation of Reliability Functions for Software) (Farr, 1991), be also utilized to provide alternate models for consideration and additional statistical information. SMERFS is available as an off-the-shelf product and is included as a reference in this thesis.

Additional research would prove very valuable if concentrated on each software metric, on the cost effectiveness of metric data requirements, or on the impacts of implemented software metrics methodologies throughout civilian and government organizations.

APPENDIX A

The set of 12 metrics as presented primarily in the Department of the Army's draft Software Test and Evaluation Procedures and Guidelines are summarized in this appendix. Three basic categories of measurement are delineated in this draft literature. Cost, schedule, computer resource utilization and software engineering environment metrics are included in a management category. Requirements traceability and stability metrics define a requirements category. Design stability, complexity, breadth of testing, depth of testing, fault profiles and reliability metrics fill out the quality category.

A. COST

During a project life-cycle, software cost data should continually be collected and analyzed. Indications of project software well-being using this cost thermometer can be expected.

Three areas require cumulative compilation. First, a scheduled baseline of budgeted funds during each project must be compiled (e.g., after 6 months the project software was expected to achieve Milestone III, and 5 million dollars were budgeted to reach this point). Next, the budgeted funds to reach actual project progress must be detailed (e.g., at 6

months the project is approaching Milestone III, and 4 million dollars were budgeted to reach this point). Finally, and perhaps most importantly, the actual funds required to reach the current project progress point must be revealed (e.g., at 6 months the project is approaching Milestone III, and 4.5 million dollars have been expended to achieve this).

Cost performance trends can be produced from these three sources of data. Variability from expected, and therefore budgeted, resource levels, as depicted in the cost (Figure 3) and cost performance (Figure 4) figures below, will provide a continual source of software management information to the software manager.

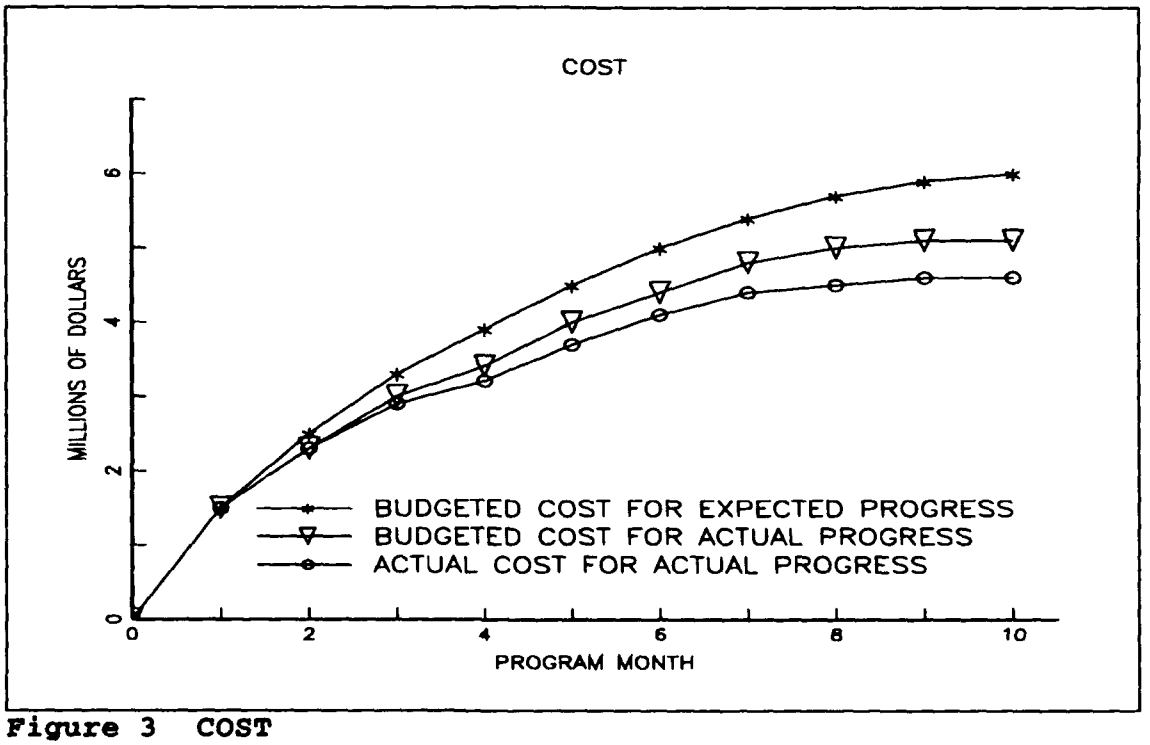

Figure 3 COST

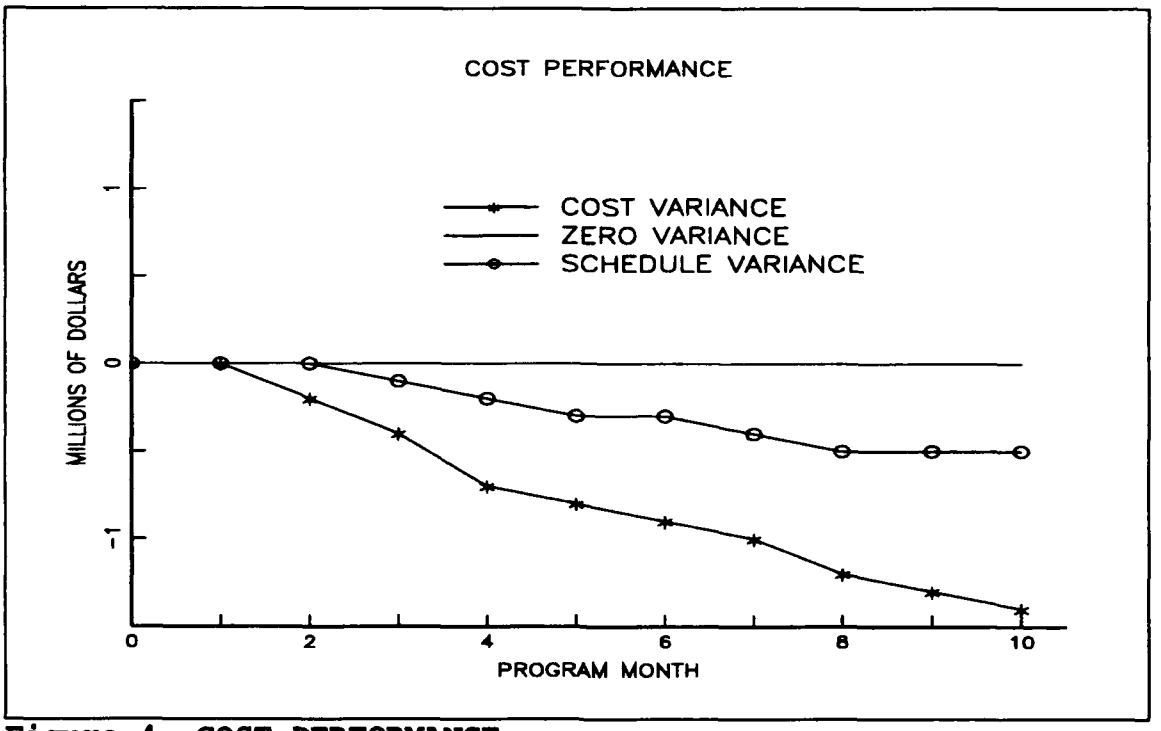

Figure 4 COST PERFORMANCE

B. SCHEDULE

Adherence to a planned schedule of milestone achievement can often not be accomplished. A schedule metric, when properly utilized, can be used as an alarm device for further investigative efforts by a software manager. Continual milestone slippage may be indicative of software problems or merely a poorly thought out schedule.

The planned month of milestone achievement should be plotted versus the current program month (e.g., at actual program month 9, Milestone IV was scheduled for completion in program month twelve). A positively sloping line, Figure 5,

would reveal milestone slippage and perhaps software trouble signs.

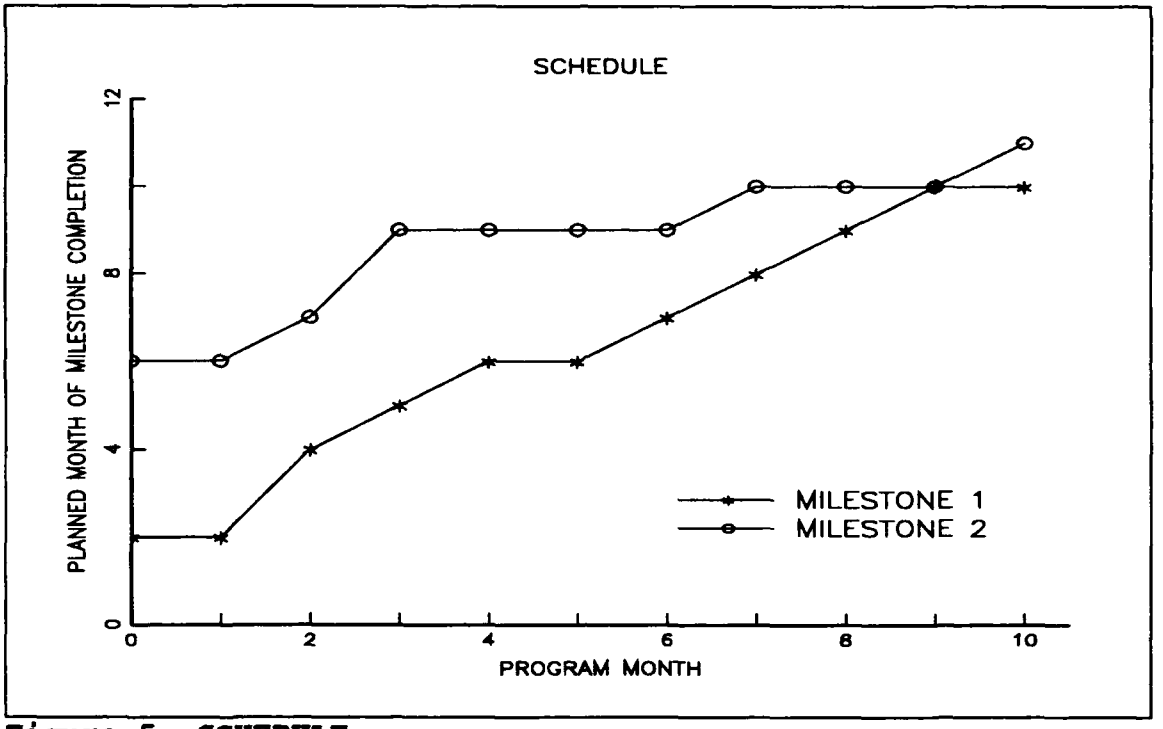

Figure 5 SCHEDULE

C. COMPUTER RESOURCE UTILIZATION

The computer resource metric portrays the degree which central processing unit capacity, memory/storage capacity and input/output capacity are changing or approaching the limits of resource availability.

The central processing unit, each input/output channel, random access memory and each mass storage device should be monitored for utilization by tracking projected usage, actual usage and target (upper bound) limits. Many off-the-shelf software packages already self-monitor utilization parameters

and some do not. In either case project software specifications must include these data collection requirements. The central processing unit capacity, Figure 6, displays a possible utilization scenario. The software manager will be able to better assess potentially critical software changes if this utilization metric were available.

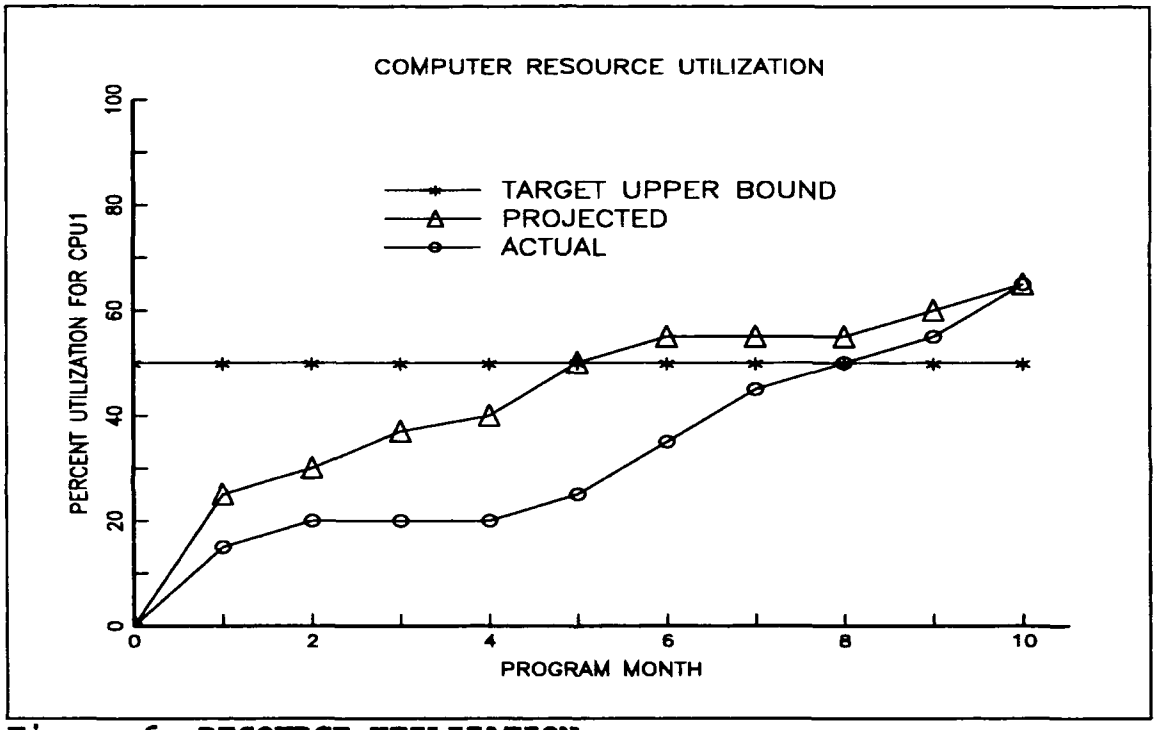

Figure 6 RESOURCE UTILIZATION

D. SOFTWARE ENGINEERING ENVIRONMENT

Contractors for software projects are varied and many. Each of these contractors operate their businesses in different manners.

The software engineering environment metric is simply a measurement of how government software contractor's

engineering practices compare to each other on a numeric rating scale. The key assumption behind this rating metric is that a quality engineering process results in a quality software product.

The data required for this engineering environment metric consists of questionnaire data and an actual assessment visit by a qualified independent group. The information from these assessments results in a numeric grade, from one to five, for each software contractor.

This engineering environment metric would be supplied to the software manager as yet another tool to help make managerial decisions concerning software progress or lack thereof.

E. REQUIREMENTS TRACEABILITY

Contractual requirements for a software package are the basis upon which development occurs. Users, administrators, software managers and testers each take great care in providing the strictest of guidance where system requirements for contractors are specified. These requirements are continually updated and improved upon.

A requirements traceability measure is essential to ensure contractor adherence to specification and their continued progress towards these requirements. This traceability measure consists of a percentage conformity index which when presented in matrix format at key milestone reviews can

provide early indications of software problems. The percentage of satisfied requirements from different sources can provide this valuable information (e.g., 86% of user requirements and 74% of the operational requirements have been met at Milestone II). Both forward and backward analysis of these data become essential as requirements are added or dropped with program maturity.

A software manager can better assess a contractor's progress and possibly gain some preventive insight for the future of the project by using these requirements traceability indicators (e.g., are the user requirements acceptable?).

F. REQUIREMENTS STABILITY

As a software package progresses towards contractual fulfillment, program managers may be forced to change any number of requirements imposed upon a contractor (e.g., while target information is being automatically updated there must be an avenue available to manually alter the number of on-line sensors). A change in the contractual goal can often be a key indicator for deeper system problems. This requirements stability metric would track the number of engineering change proposals, the percent source line of code changed and the percentage of software modules affected by any change. Early indications can be obtained, both good and bad, pertaining to overall software well-being. A requirements stability scenario is depicted in Figure 7.

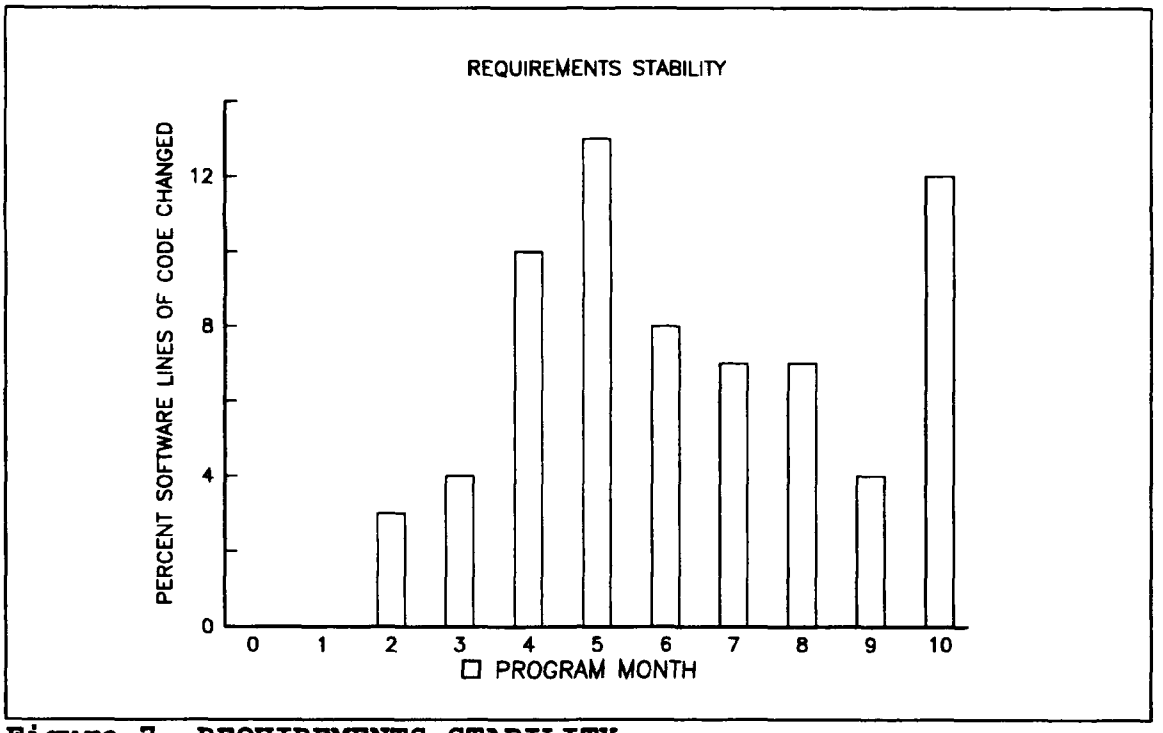

Figure 7 REQUIREMENTS STABILITY

G. DESIGN STABILITY

The design stability metric is used to depict the amount of changes made to the design of software. This metric should be utilized throughout the life of a project. The stability is defined by ((M-(Fa+Fc+Fd)) / M) and the design progress ratio is defined by (M / T). M is the number of modules within the software. Fc is the number of modules that include design related changes since the latest stability check. Fa is the number of modules added since the latest stability check. Fd is the number of modules deleted since the last stability check. T is the number of modules projected for the project. The design stability equation is simply a

mathematical representation of the degree that software design elements have to be changed as the project progresses.

H. COMPLEXITY

Complexity analysis is based on a single assumption; the more complex the software, the harder it is to test and maintain.

The McCabe cyclomatic complexity metric should be collected and analyzed throughout a project's lifetime. Each module should be analyzed with McCabe's metric. The complexity in McCabe's metric equals (E - N + 2P), where E equals the number of edges (e.g., software calls or branches to procedures or functions), N equals the number of nodes (e.g. software procedures or functions) and P equals the number of stand alone components (e.g., within a module, no branches between one set of nodes and any other).

An analysis of complexity for the overall software package using McCabe analysis consists of counting all modules whose complexity rating falls in a certain interval and displaying these results for all modules. An example of this technique is shown in Figure 8.

Other complexity measures like the Halstead approach are available and thoroughly discussed in the Army's Software Test and Evaluation Guidelines. As a result of diligent complexity analysis, the program manager should gain insight into future effort concentrations for test and development of the

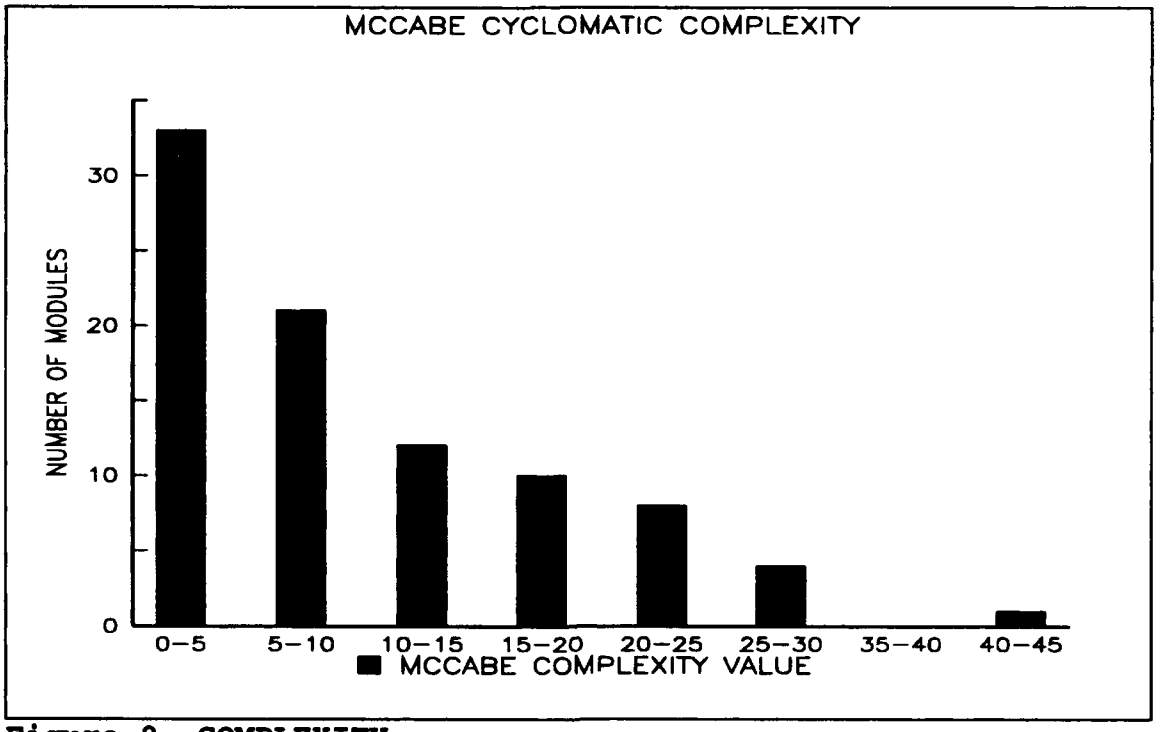

Figure 8 COMPLEXITY

software.

I. BREADTH OF TESTING

The breadth metric details both how well a software package demonstrates required functionality and how much testing has been performed.

Three measurement ratios should be recorded and analyzed throughout the lifetime of a project; test coverage which equals the number of requirements tested over the total number of requirements, test success which equals the number of requirements passed over the number of requirements tested and overall success which equals the number of requirements passed

over the total number of requirements. Figure 9 displays a plausible breadth of test scenario.

Insight into testing adequacy and concentration can

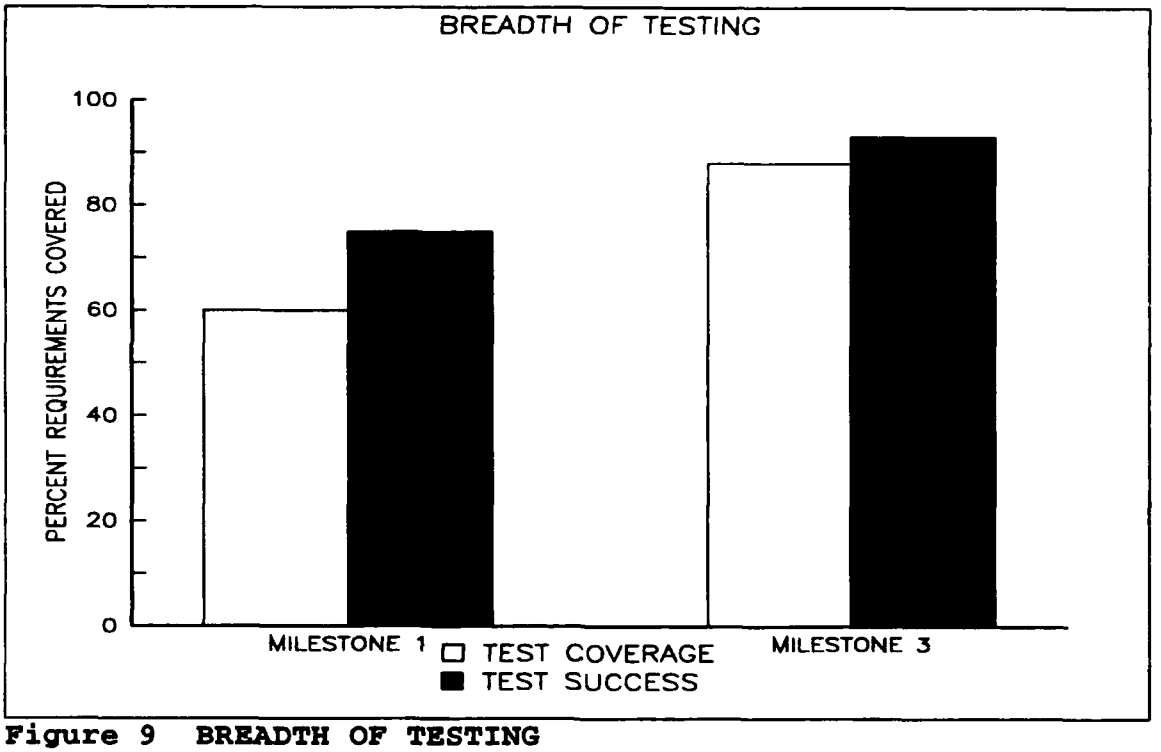

Figure 9 BREADTH OF TESTING

readily be made by a program manager with this breadth metric, when analyzed in conjunction with other metrics.

J. DEPTH OF TESTING

The depth metric details the extent and success of testing on a software intensive system. Many execution paths exist within any software package. If the testing process is not exploring a wide range of these executable paths, then undisclosed difficulties may present themselves later in the system lifecycle.

Depth is comprised of three separate measurement ratios (path, statement and domain) which are tracked for each module in the system. The path ratio is the number of paths in a module that have been successfully executed at least once over the total number of paths in the module. The statement measurement ratio is the number of executable statements in a module that have been successfully executed at least once over the total number of executable statements in the module. The domain measurement ratio is simply the number of various input combinations attempted over the total number of available input instances. Figure 10 depicts a plausible depth of testing scenario utilizing the path ratio.

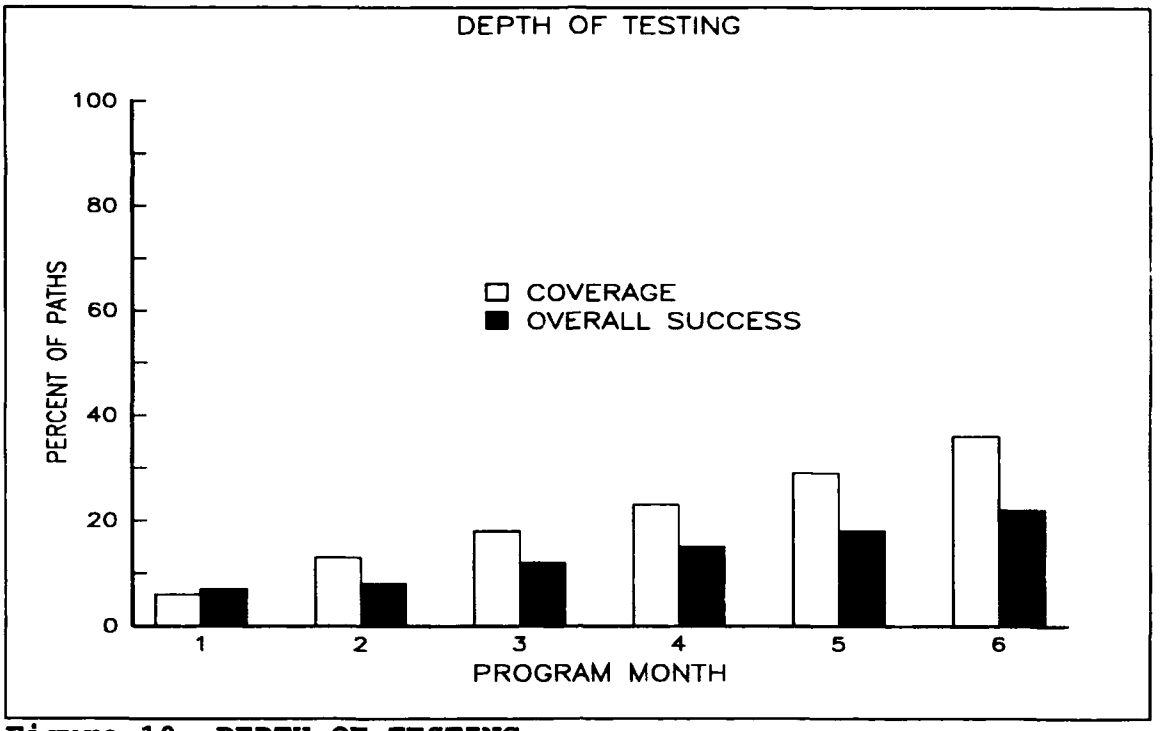

Figure 10 DEPTH OF TESTING

Automated (internal) data collection is necessary for the path and statement ratios. To insure this automated process is included in the software capabilities, these data collection functions must be included in the software system requirements.

K. FAULT PROFILE

This metric is used to provide insight into software quality and the contractor's ability to correct known faults within the software. A simple plot of the cumulative number of software trouble reports will reveal insight into the software readiness, Figure 11.

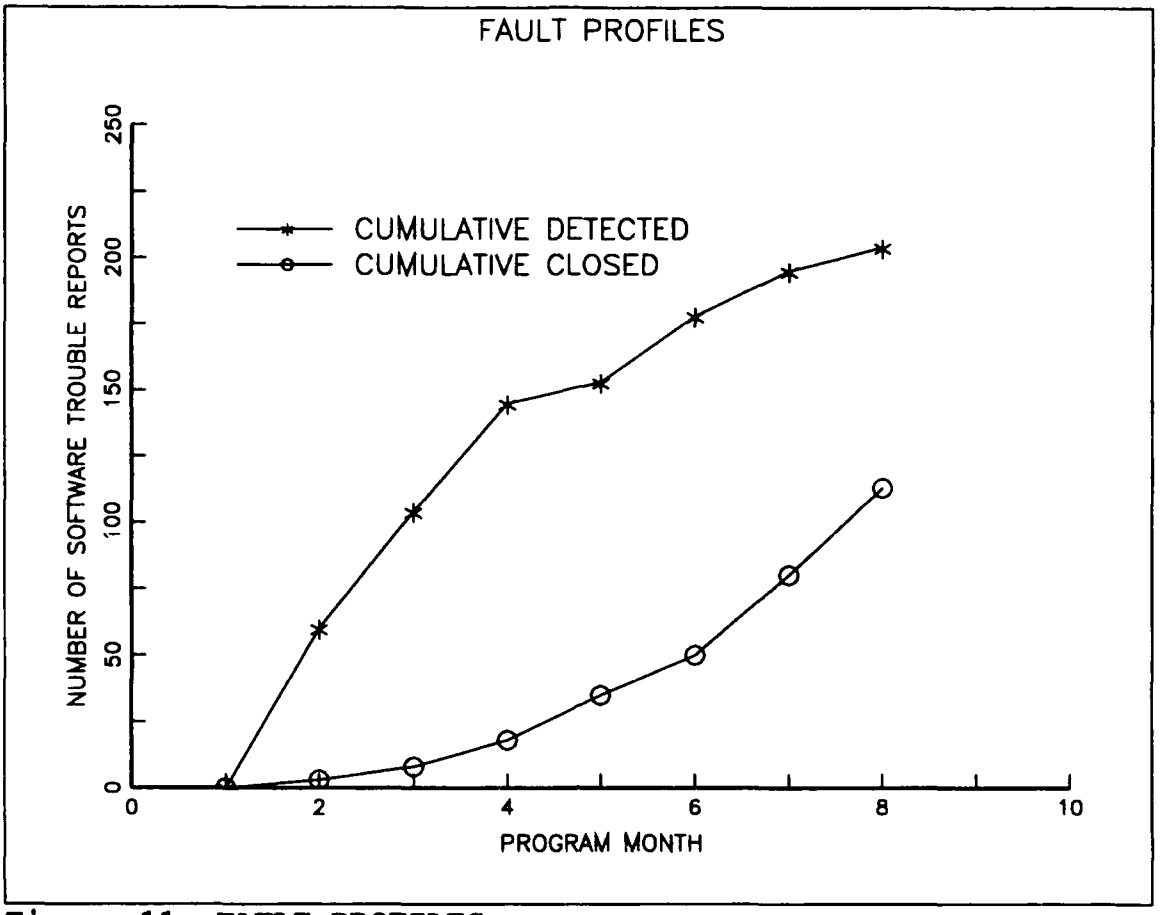

Figure 11 FAULT PROFILES

Next, a plot of fault age (i.e., time a fault remains non-corrected or open) is a very useful fault profile tool, Figure 12.

L. RELIABILITY

Reliability in software has been studied from many different angles and each of these angles have their own positive points. Every software intensive project should require a complete analysis when determining which reliability technique(s) best fit a particular system. Reliability models

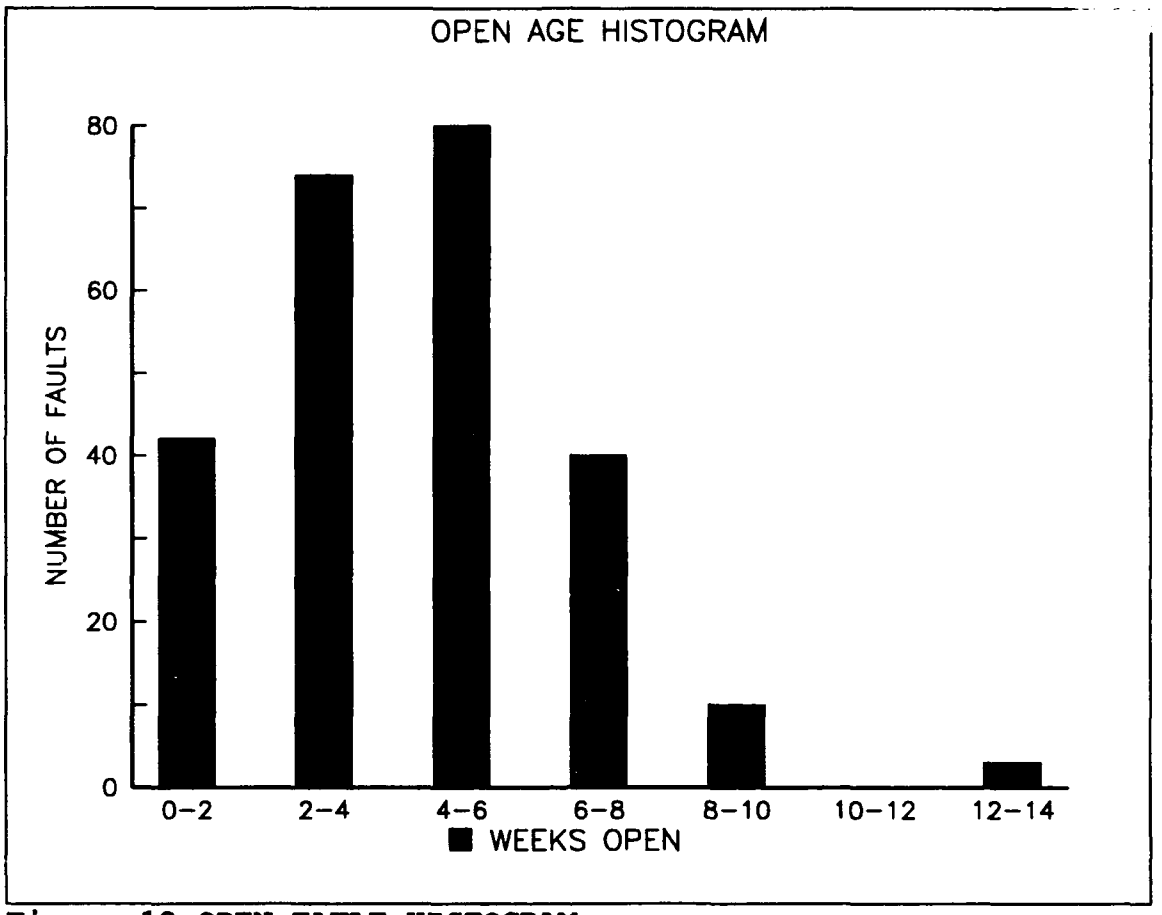

Figure 12 OPEN FAULT HISTOGRAM

can be used as predictive tools just as our model in Chapter III. SMERFS is an example of a software reliability modeling tool that could be utilized as an Army reliability metric. The reliability metric consists primarily of fault analysis models like the one developed in this thesis. With the use of an applicable reliability fault model resource savings can be expected as with any applicable software metric.

APPENDIX B

In this appendix, the NHPP with independent time interval lengths, discussed in Chapter III is represented in an algorithm. This algorithm was used to generate the results referred to in Chapter IV. The program was written in Pascal for personal computer use and was executed utilizing Turbo Pascal 6.0 (Borland, 1990), a commercial Pascal package.

```
PROGRAM SoftwareReliability (cumfail,output);

{Software Reliability}
{Software cumulative fault count analysis with bootstrapping}
{LT Doug Burton, USN}
{Naval Postgraduate School, Monterey, CA  93943}

CONST  epsilon = 1.0E-03;
       LengthofDataSet = 10;
       iteration = 200;

TYPE   DATA = ARRAY [0..LengthofDataSet] of real;
       RUNS = ARRAY [0..iteration] of real;

VAR    Faults,Times,FaultsEnd,LAMi,tempfaults :DATA;
       lamboot,muboot,lamprob,muprob,mean :RUNS;

       XFirst,MU,LAMBDA,TotalTime,TotalFaults:real;
       TotItFaults,A,B,key,Tother,percentile :real;

       i,j,k,low,high :integer;

       Cumfail :text;

{Faults       = Number of system faults in each period}
{Times        = Cumulative execution time for each period}
{FaultsEnd    = Cumulative system faults for each period}
{LAMi         = Set of Lambdas for each period}
{tempfaults   = Temporary random fault counts in each period}
{lamboot      = Lambdas generated from each bootstrapping run}
{muboot       = MUs generated from each bootstrapping run}
{mean         = Calculated means for each bootstrap iteration}
```

```
{XFirst      = Initial X value for EXP(-MU*total system time)}
{MU           = Model parameter}
{LAMBDA       = Model Parameter}
{TotalTime    = Total system execution time}
{TotalFaults = Total system faults encountered}
{TotItFaults = Total system faults for a bootstrap iteration}
{A and B      = Dummy variables}
{key          = Place holding variable}
{T            = input variable for additional time analysis}
{other        = a yes, no variable}
{percentile   = user requested confidence percentile}
{i and j       = Counters}
{low and high = The 5th and 95th percentiles}
{Cumfail       = The cumulative data file being analyzed}


{*********FUNCTION TO DETERMINE THE FIRST X VALUE*********}

FUNCTION FirstXValue: real;

VAR  SearchValue,FirstHalfCount,SecondHalfCount: real;
     i :integer;

{SearchValue      = Half of the total run time}
{FirstHalfCount   = Faults in the first half of the run time}
{SecondHalfCount = Faults in the last half of the run time}
{i                = location memory value}

BEGIN
   SearchValue:= (TotalTime/2);
   FirstHalfCount:= 0;
   SecondHalfCount:= 0;
   i:= 1;

   REPEAT
      i:=(i+1);
   UNTIL (Times[i] >= SearchValue);

   FirstHalfCount:= (FaultsEnd[i-1]+
              ((FaultsEnd[i]-FaultsEnd[i-1])*

((SearchValue-Times[i-1])/(Times[i]-Times[i-1]))));

         S e c o n d H a l f C o u n t : =
(FaultsEnd[LengthofDataSet]-FirstHalfCount);

FirstXValue:= ((SecondHalfCount/FirstHalfCount)*
                        (SecondHalfCount/FirstHalfCount));

END; {FirstXValue}
```

```
{***************FUNCTION TO SOLVE FOR MU******************}

FUNCTION FindMU (XOne,TFaults :real; F :DATA):real;

VAR  MUCurrent,MUNew,s,x,t1,t2,t3,t4 :real;
     i :integer;

{MUCurrent    = Last estimate for MU}
{MUNew        = This estimate for MU}
{s and x      = Iteration Values}
{t1,t2,t3,t4 = temporary variables}
{i            = counter}

BEGIN
   s:= 0;
   x:= 0;
   i:= 0;

   t1:= 0;
   t2:= 0;
   t3:= 0;
   t4:= 0;

   MUCurrent:= (((-1)*LN(XOne))/TotalTime);
   MUNew:= MUCurrent;

   REPEAT
      MUCurrent:=MUNew;

      FOR i:= 1 TO LengthofDataSet DO BEGIN

          t1:=Times[i]*EXP((-1)*MUCurrent*Times[i]);

          t2:=Times[i-1]*EXP((-1)*MUCurrent*Times[i-1]);

          t3:=EXP((-1)*MUCurrent*Times[i-1])
                        -EXP((-1)*MUCurrent*Times[i]);

          t4:=t4 + (F[i]*((t1-t2)/t3));

      END; {FOR Loop}

      IF (t4 < (0.01)) THEN t4:= 0.01;

      s:= (t4/(TFaults*TotalTime));
      x:= (s/(1+s));
      t4:= 0;

      MUNew:= (((-1)*LN(X))/TotalTime);

   UNTIL ((ABS(MUNew-MUCurrent)) <= epsilon);
```

```
FindMU:= MUCurrent;

END; {FindMU}

{********PROCEDURE TO GET LAMBDAS FOR EACH PERIOD*********}

PROCEDURE GenerateLambdas;

BEGIN
   FOR i:= 2 TO LengthofDataSet DO BEGIN

LAMi[i]:=(LAMBDA*(EXP(-1*MU*Times[i-1])-EXP(-1*MU*Times[i]))
);
   END; {For loop}

   LAMi[1]:=(LAMBDA*(1-EXP(-1*MU*Times[1])));
END; {Procedure GenerateLambdas}

{******PROCEDURE TO ASSIGN POISSON RV'S TO TEMPFAULTS*****}

PROCEDURE AssignPoissons;

VAR A,B,W,X,U :real;

BEGIN
   FOR i:= 1 TO LengthofDataSet DO BEGIN
      W:= EXP((-1)*LAMi[i]);
      X:= 0;
      A:= W;
      B:= A;

      U:= RANDOM(1000)/1000;

      WHILE (U>A) AND (A<=0.9999) AND (B>=0.0001) DO BEGIN
         X:=X+1;
         B:= (B*LAMi[i])/X;
         A:= A+B;
      END; {While}

      tempfaults[i]:=X;
      TotItFaults:= TotItFaults+tempfaults[i];

   END; {FOR Loop}

END; {PROCEDURE AssignPoissons}

{********************THE MAIN PROGRAM**********************}

BEGIN
   XFirst:= 0;
   MU:= 0;
```

```
   LAMBDA:= 0;
   TotalTime:= 0;
   TotalFaults:= 0;
   TotItFaults:= 0;
   T:= 0;
   i:= 0;
   j:= 0;
   k:= 0;

   FOR i:= 0 TO iteration DO BEGIN
      lamboot[i]:= 0;
      muboot[i]:= 0;
      mean[i]:= 0;
   END; {FOR Loop}

   FOR i:= 0 TO LengthofDataSet DO BEGIN
      Times[i]:= 0;
      Faults[i]:= 0;
      FaultsEnd[i]:= 0;
      LAMi[i]:= 0;
      tempfaults[i]:= 0;
   END; {FOR Loop}

{*************INPUT THE DATA*****************}

{Data comes from a data file listed next to the ASSIGN command
below.}

{The data in this file must be in a two column format.}

{The first column will contain the cumulative system execution
times.}

{The second column will contain the cumulative system fault
counts that correspond to the times of column one.}

{Hard returns should be used to place the data into this
file.}

{No column headers are necessary and will halt the program if
used.}

ASSIGN(Cumfail,'yourfile.dat');

RESET (Cumfail);

WHILE NOT EOF (Cumfail) DO BEGIN
   j:=(j+1);
   READLN(Cumfail,A,B);
   Times[j]:= A;
   FaultsEnd[j]:= B;
```

```
END; {WHILE}

FOR i:= 1 TO LengthofDataSet DO BEGIN
   Faults[i]:=FaultsEnd[i]-FaultsEnd[i-1];
END; {FOR Loop}

TotalTime:= Times[LengthofDataSet];
TotalFaults:= FaultsEnd[LengthofDataSet];

{***************Function and Procedure calls**************}

RANDOMIZE;

XFirst:= FirstXValue;

MU:= FindMU(XFirst,TotalFaults,Faults);

LAMBDA:=(TotalFaults/(1-EXP((-1)*MU*TotalTime)));

GenerateLambdas;

FOR k:= 1 TO iteration DO BEGIN
   TotItFaults:= 0;
   AssignPoissons;
   muboot[k]:=FindMU(XFirst,TotItFaults,tempfaults);

lamboot[k]:=(TotItFaults/(1-EXP((-1)*muboot[k]*TotalTime)));
END; {FOR Loop}

low:= ROUND(iteration * (0.05));
high:= ROUND(iteration * (0.95));


Writeln('Please input the amount of time you want');
Writeln('to examine for faults in the future!');
Writeln('Any positive number will do.');
Writeln('CAUTION!!!');
Writeln('This amount of time MUST have the same');
Writeln('time units as your original data');
Writeln('EXAMPLE--Data is in minutes therefore your');
Writeln('input is 20.5 "minutes".');
Writeln('e.g. The upcoming test requires 20.5 minutes of');
Writeln('software execution time.');

Readln(T);

Writeln;

FOR i:= 1 TO iteration DO BEGIN
   mean[i]:= lamboot[i]*
             (EXP((-1)*muboot[i]*Times[LengthofDataSet]))*
```

```
                (1-EXP((-1)*muboot[i]*T));
END;

FOR j:= 2 TO iteration DO BEGIN
   key:= mean[j];
   i:= (j-1);

   WHILE ((i > 0) AND (mean[i] > key)) DO BEGIN
      mean[i+1]:=mean[i];
      i:=(i-1);
      mean[i+1]:=key;
   END;

END;
Writeln('WITH 95% CONFIDENCE!');
Writeln('The maximum and minimum number of expected faults
in');
Writeln('the next ', T:3:3,' time units is');
Writeln(mean[high]:4:4,'    and    ',mean[low]:4:4,'
respectively.');

Writeln;
Writeln;

Writeln('Do you want to calculate a different confidence
bound?');
Writeln('e.g. Instead of 95%, you would like 98% bounds.');
Writeln;
Writeln('If yes, input a one now.  If no, input anything
else.');

Readln(other);

Writeln;


IF (other=1) THEN BEGIN

Writeln('Input your desired confidence percentile now.');
Writeln('Use decimal format--e.g. 0.90 is your input for
90%');

Readln(percentile);

Writeln;

low:= ROUND(iteration * (1-percentile));
high:= ROUND(iteration * percentile);

Writeln('WITH ',(percentile*100):2:2,' % CONFIDENCE!');
```

```
Writeln('The maximum and minimum number of expected faults
in');
Writeln('the next ',T:3:3,' time units is');
Writeln(mean[high]:4:4,'    and    ',mean[low]:4:4,'
respectively.');

END;

CLOSE (Cumfail);

END.
```

APPENDIX C

Suppose that the precise inter-occurrence run-times of failures are recorded: let t_1 be the time from the moment the program begins running until the first fault is encountered; $\overline{t}_{j-1}$ is the cumulative run time until the (j-1)st fault; $\overline{t}_j=\overline{t}_{j-1}+x_j$, so x_j is the observed run time between the time of occurrence of fault j-1 and fault j.

Given that the time to the (j-1)st fault discovery has been observed, namely $\overline{t}_{j-1}=x_1+x_2+x_3+\ldots+x_{j-1}$, the probability that the time to the next (jth) fault exceeds x is, according to our model,

$$P(X_j>x|X_1+X_2+\ldots+X_{j-1}=\overline{t}_{j-1})=\frac{e^{-\lambda\cdot[1-e^{-\mu\cdot(\overline{t}_{j-1}+x)}]}}{e^{-\lambda\cdot[1-e^{-\mu\cdot\overline{t}_{j-1}}]}} \qquad (C.1)$$

and then

$$P(X_j>x|X_1+X_2+\ldots+X_{j-1}=\overline{t}_{j-1})=e^{-\lambda\cdot e^{-\mu\cdot\overline{t}_{j-1}}\cdot(1-e^{-\mu\cdot x})} \qquad (C.2)$$

where X_j (j=1,2,3,...) represents the random inter-occurrence time variables in our non-homogeneous Poisson process model. Differentiation with respect to x gives the density

$$f_j(x;\overline{t}_{j-1})=\mu\cdot\lambda\cdot(e^{-\lambda\cdot e^{-\mu\cdot\overline{t}_{j-1}}\cdot(1-e^{-\mu\cdot x})})\cdot(e^{-\mu\cdot\overline{t}_{j-1}})\cdot(e^{-\mu\cdot x}). \qquad (C.3)$$

Now, if we observe $X_j=x_j$ then the likelihood component becomes $f_j(x_j;\bar{t}_{j-1})$. Again, with $\bar{t}_j$ representing our new cumulative time of fault j occurrence and J being the number of the last fault,

$$L(\lambda,\mu|data)=\prod_{j=1}^{J}\lambda\cdot\mu\cdot(e^{-\lambda\cdot e^{-\mu\cdot\bar{t}_{j-1}}\cdot(1-e^{-\mu\cdot x_j})})\cdot(e^{-\mu\cdot\bar{t}_{j-1}})\cdot(e^{-\mu\cdot x_j}) \quad (C.4)$$

and taking the natural log of this likelihood function results in

$$\bar{L}(\lambda,\mu|data)=J\cdot\ln\lambda+J\cdot\ln\mu+[-\lambda\cdot\sum_{j-1}^{J}(e^{-\mu\cdot\bar{t}_{j-1}}-e^{-\mu\cdot\bar{t}_j})]-[\mu\cdot\sum_{j=1}^{J}\bar{t}_j]. \quad (C.5)$$

Differentiation with respect to λ gives

$$\frac{d\bar{L}}{d\lambda}=\frac{J}{\lambda}-\sum_{j=1}^{J}[e^{-\mu\cdot\bar{t}_{j-1}}-e^{-\mu\cdot\bar{t}_j}]. \quad (C.6)$$

Setting the derivative equal to zero and solving gives

$$\hat{\lambda}=\frac{1}{[(1-e^{-\mu\cdot\bar{t}_J})/J]}, \quad (C.7)$$

which resembles (9) in Chapter III, but is not the same since the times differ.

Next, differentiation with respect to μ gives

$$\frac{d\overline{L}}{d\mu} = \frac{J}{\mu} - (\lambda \cdot \sum_{j=1}^{J} [\,(\overline{t}_j \cdot e^{-\mu \cdot \overline{t}_j}) - (\overline{t}_{j-1} \cdot e^{-\mu \cdot \overline{t}_{j-1}})\,]) - \sum_{j=1}^{J} \overline{t}_j . \qquad (C.8)$$

If the above derivative is equated to zero an equation for μ is obtained, once substitution of the expression for λ in terms of μ, as (9) in Chapter III, is carried out. An iterative process for μ could use this formula:

$$\hat{\mu}_{n+1} = \frac{J}{\hat{\lambda} \cdot \sum_{j=1}^{J} ([\overline{t}_j \cdot e^{-\mu_n \cdot \overline{t}_j} - \overline{t}_{j-1} \cdot e^{-\mu_n \cdot \overline{t}_{j-1}}] + \overline{t}_j)} . \qquad (C.9)$$

An iterative process for both $\hat{\lambda}$ and $\hat{\mu}$ would result in an estimate for the model parameters μ and λ. With these estimates in hand, Bootstrapping can then be applied to produce confidence limits for reliability predictions as in the previous case.

LIST OF REFERENCES

Borland International, *Turbo Pascal 6.0*, 1990.

Commander, Operational Test and Evaluation Forces, (COMOPTEVFOR), Software-Intensive System Data, 1992.

Dennison, T., "Fitting and Prediction Uncertainty for a Software Reliability Model," *Master's Thesis*, Naval Postgraduate School, 1992.

Draft Recommended Practice for Software Reliability, American Institute of Aeronautics and Astronautics, April, 1992.

Standard for a Software Quality Metrics Methodology, Institute of Electrical and Electronics Engineers, Inc., 1993.

Draft Software Test and Evaluation Guidelines, Headquarters Department of the Army, Vol.6, June, 1992.

Efron, B., "Bootstrap Confidence Intervals," *Biometrika*, Vol.72, No.1, April, 1985.

Farr W. and Smith O., *Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS)*, Rev.2, March 1991.

Fishman, G., *Principles of Discrete Event Simulation*, pp.440-441, Wiley, 1978.

Goel, and Okomoto, "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Transactions on Reliability*, Vol.R-28, No.3, pp.206-211, 1979.

GRAFSTAT, AN APL System for Interactive Scientific-Engineering Plotting, Data Analysis, Applied Statistics, and Graphics Output Development, IBM, 1988.

Keller, T., Case Study-Predicting, Controlling, and Assessing Reliability for Space Shuttle Primary Avionics Software, IBM, 1992.

Larson, H., *Introduction to Probability Theory and Statistical Inference*, pp.361-362, Wiley, 1982.

McCabe, T., "Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, Number 4, pp.308-320, December 1976.

Musa, J., Tools for Measuring Software Reliability," *IEEE Spectrum*, February, 1989.

Schneidewind, N., "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Engineering*, Vol.SE-5, No.3, May, 1979.

Schneidewind N. and Keller T., "Applying Reliability Models to the Space Shuttle," *IEEE Software*, July, 1992.

Schneidewind, N., "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering*, Vol.18, No.5, May, 1992.

Siefert, D., "Implementing Software Reliability Measures," *The NCR Journal*, Vol.3, No.1, March, 1989.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 052 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Professor Donald P. Gaver Department of Operations Research (OR/Ga) Naval Postgraduate School Monterey, CA 93943-5002	1
4.	Professor Norman F. Schneidewind Department of Administrative Sciences (AS/Sc) Naval Postgraduate School Monterey, CA 93943-5002	1
5.	Professor Patricia Jacobs Department of Operations Research (OR/Ja) Naval Postgraduate School Monterey, CA 93943-5002	1
6.	Commander, Operational Test and Evaluation Force Chief of Staff Norfolk, VA 23511	1
7.	Commander, Operational Test and Evaluation Force Technical Director, Code OOT Norfolk, VA 23511	1
8.	Commander, Operational Test and Evaluation Force Deputy Chief of Staff for OTE Support, 30 Division Attention: Code 333 Norfolk, VA 23511	2
9.	LT Douglas R. Burton 7843 Windy Lane Massillon, OH 44646	1